Tidy Data (part 2)

Contents

1 Review

So far we have:

- learned how to load in packages.
- learned how to create some plots with built in datasets.
- Some basics of coding.
- General principals of calling functions.
- Transform data.
- Read in data from various sources.
- Reading in data.
- Tidying and Pivoting

Overview of the class:

- Finish up with tidying:
  - separating and uniting data.
  - dealing with missing data.

Recall the tables from last class:

```
library(tidyverse)
table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
```

```
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```
# Spread across two tibbles
table4a # cases
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
table4b # population
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

2 Separating and uniting

So far you've learned how to tidy `table2` and `table4`, but not `table3`. `table3` has a different problem: we have one column (rate) that contains two variables (cases and population). To fix this problem, we'll need the `separate()` function. You'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

2.1 Separate

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `table3`:

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

The rate column contains both cases and population variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into different columns.

```
table3 |>
  separate(rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

By default, `separate()` will split values wherever it sees a *non-alphanumeric character* (i.e. a character that isn't a number or letter). For example, in the code above, `separate()` split the values of rate at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the sep argument of `separate()`. For example, we could rewrite the code above as:

```
table3 |>
  separate(rate, into = c("cases", "population"), sep = "/")
```

```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Look carefully at the column types: you'll notice that cases and population are character columns. This is the default behaviour in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

```
table3 |>
  separate(rate, into = c("cases", "population"), convert = TRUE)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
```

```
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

You can also pass a vector of integers to sep. separate() will interpret the integers as positions to split at. You can use this arrangement to separate the last two digits of each year. This make this data less tidy, but is useful in other cases, as you'll see in a little bit.

```
table5 <- table3 |>
  separate(rate, into = c("cases", "population"), convert = TRUE) |>
  separate(year, into = c("century", "year"), sep = 2)
table5
```

```
## # A tibble: 6 x 5
##   country     century year   cases population
##   <chr>       <chr>   <chr>  <int>      <int>
## 1 Afghanistan 19      99       745   19987071
## 2 Afghanistan 20      00      2666   20595360
## 3 Brazil      19      99     37737  172006362
## 4 Brazil      20      00     80488  174504898
## 5 China       19      99    212258 1272915272
## 6 China       20      00    213766 1280428583
```

2.2 Unite

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

We can use `unite()` to rejoin the century and year columns that we created in the last example. That data is saved as tidyr::table5. unite() takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in dplyr::select() style:

```
table5 |>
  unite(new, century, year)
```

```
## # A tibble: 6 x 4
##   country     new    cases population
##   <chr>       <chr>  <int>      <int>
## 1 Afghanistan 19_99    745   19987071
## 2 Afghanistan 20_00   2666   20595360
## 3 Brazil      19_99  37737  172006362
## 4 Brazil      20_00  80488  174504898
## 5 China       19_99 212258 1272915272
## 6 China       20_00 213766 1280428583
```

In this case we also need to use the sep argument. The default will place an underscore (_) between the values from different columns. Here we don't want any separator so we use " ":

```
table5 |>
  unite(new, century, year, sep = "")
```

```
## # A tibble: 6 x 4
##   country     new    cases population
##   <chr>       <chr>  <int>      <int>
## 1 Afghanistan 1999     745   19987071
## 2 Afghanistan 2000    2666   20595360
```

```
## 3 Brazil      1999   37737  172006362
## 4 Brazil      2000   80488  174504898
## 5 China       1999  212258 1272915272
## 6 China       2000  213766 1280428583
```

2.3 Recreating the variable groups

Previously we changed the column names before using `pivot_longer()` with the `who` dataset so that the groups would be separated. This time, we're going to repeat that exercise, using the separate function. Recall, the data looks like.

```
who_orig <- tidyr::who
who_orig
```

```
## # A tibble: 7,240 x 60
##    country  iso2  iso3   year new_sp_m014 new_sp_m1524 new_sp_m2534 new_sp_m3544
##    <chr>    <chr> <chr> <int>       <int>        <int>        <int>        <int>
##  1 Afghani~ AF    AFG    1980          NA           NA           NA           NA
##  2 Afghani~ AF    AFG    1981          NA           NA           NA           NA
##  3 Afghani~ AF    AFG    1982          NA           NA           NA           NA
##  4 Afghani~ AF    AFG    1983          NA           NA           NA           NA
##  5 Afghani~ AF    AFG    1984          NA           NA           NA           NA
##  6 Afghani~ AF    AFG    1985          NA           NA           NA           NA
##  7 Afghani~ AF    AFG    1986          NA           NA           NA           NA
##  8 Afghani~ AF    AFG    1987          NA           NA           NA           NA
##  9 Afghani~ AF    AFG    1988          NA           NA           NA           NA
## 10 Afghani~ AF    AFG    1989          NA           NA           NA           NA
## # ... with 7,230 more rows, and 52 more variables: new_sp_m4554 <int>,
## #   new_sp_m5564 <int>, new_sp_m65 <int>, new_sp_f014 <int>,
## #   new_sp_f1524 <int>, new_sp_f2534 <int>, new_sp_f3544 <int>,
## #   new_sp_f4554 <int>, new_sp_f5564 <int>, new_sp_f65 <int>,
## #   new_sn_m014 <int>, new_sn_m1524 <int>, new_sn_m2534 <int>,
## #   new_sn_m3544 <int>, new_sn_m4554 <int>, new_sn_m5564 <int>,
## #   new_sn_m65 <int>, new_sn_f014 <int>, new_sn_f1524 <int>, ...
```

This time, we'll first use `pivot_longer` and put the variable names into a variable called `key`

```
who1 <- who_orig |>
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_to = "key",
    values_to = "cases",
    values_drop_na = TRUE
  )
who1
```

```
## # A tibble: 76,046 x 6
##    country     iso2  iso3   year key          cases
##    <chr>       <chr> <chr> <int> <chr>        <int>
##  1 Afghanistan AF    AFG    1997 new_sp_m014      0
##  2 Afghanistan AF    AFG    1997 new_sp_m1524    10
##  3 Afghanistan AF    AFG    1997 new_sp_m2534     6
##  4 Afghanistan AF    AFG    1997 new_sp_m3544     3
##  5 Afghanistan AF    AFG    1997 new_sp_m4554     5
##  6 Afghanistan AF    AFG    1997 new_sp_m5564     2
##  7 Afghanistan AF    AFG    1997 new_sp_m65       0
```

```
##  8 Afghanistan AF    AFG    1997 new_sp_f014      5
##  9 Afghanistan AF    AFG    1997 new_sp_f1524    38
## 10 Afghanistan AF    AFG    1997 new_sp_f2534    36
## # ... with 76,036 more rows
```

We need to make a minor fix to the format of the column names: unfortunately the names are slightly inconsistent because instead of `new_rel` we have `newrel` (we saw this previously).

```
who2 <- who1 |>
  mutate(key = stringr::str_replace(key, "newrel", "new_rel"))
unique(who2$key)
```

```
##  [1] "new_sp_m014"   "new_sp_m1524"  "new_sp_m2534"  "new_sp_m3544"
##  [5] "new_sp_m4554"  "new_sp_m5564"  "new_sp_m65"    "new_sp_f014"
##  [9] "new_sp_f1524"  "new_sp_f2534"  "new_sp_f3544"  "new_sp_f4554"
## [13] "new_sp_f5564"  "new_sp_f65"    "new_sn_m014"   "new_sn_m1524"
## [17] "new_sn_m2534"  "new_sn_m3544"  "new_sn_m4554"  "new_sn_m5564"
## [21] "new_sn_m65"    "new_ep_m014"   "new_ep_m1524"  "new_ep_m2534"
## [25] "new_ep_m3544"  "new_ep_m4554"  "new_ep_m5564"  "new_ep_m65"
## [29] "new_sn_f014"   "new_rel_m014"  "new_rel_f014"  "new_sn_f1524"
## [33] "new_sn_f2534"  "new_sn_f3544"  "new_sn_f4554"  "new_sn_f5564"
## [37] "new_sn_f65"    "new_ep_f014"   "new_ep_f1524"  "new_ep_f2534"
## [41] "new_ep_f3544"  "new_ep_f4554"  "new_ep_f5564"  "new_ep_f65"
## [45] "new_rel_m1524" "new_rel_m2534" "new_rel_m3544" "new_rel_m4554"
## [49] "new_rel_m5564" "new_rel_m65"   "new_rel_f1524" "new_rel_f2534"
## [53] "new_rel_f3544" "new_rel_f4554" "new_rel_f5564" "new_rel_f65"
```

We can separate the values in each code with two passes of separate(). The first pass will split the codes at each underscore.

```
who3 <- who2 |>
  separate(key, c("new", "type", "sexage"), sep = "_")
who3
```

```
## # A tibble: 76,046 x 8
##    country     iso2  iso3   year new   type  sexage cases
##    <chr>       <chr> <chr> <int> <chr> <chr> <chr>  <int>
##  1 Afghanistan AF    AFG    1997 new   sp    m014       0
##  2 Afghanistan AF    AFG    1997 new   sp    m1524     10
##  3 Afghanistan AF    AFG    1997 new   sp    m2534      6
##  4 Afghanistan AF    AFG    1997 new   sp    m3544      3
##  5 Afghanistan AF    AFG    1997 new   sp    m4554      5
##  6 Afghanistan AF    AFG    1997 new   sp    m5564      2
##  7 Afghanistan AF    AFG    1997 new   sp    m65        0
##  8 Afghanistan AF    AFG    1997 new   sp    f014       5
##  9 Afghanistan AF    AFG    1997 new   sp    f1524     38
## 10 Afghanistan AF    AFG    1997 new   sp    f2534     36
## # ... with 76,036 more rows
```

Next we'll separate sexage into sex and age by splitting after the first character:

```
who4 <- who3 |>
  separate(sexage, c("sex", "age"), sep = 1)
who4
```

```
## # A tibble: 76,046 x 9
##    country     iso2  iso3   year new   type  sex   age   cases
##    <chr>       <chr> <chr> <int> <chr> <chr> <chr> <chr> <int>
```

```
##  1 Afghanistan AF    AFG    1997 new   sp    m     014       0
##  2 Afghanistan AF    AFG    1997 new   sp    m     1524     10
##  3 Afghanistan AF    AFG    1997 new   sp    m     2534      6
##  4 Afghanistan AF    AFG    1997 new   sp    m     3544      3
##  5 Afghanistan AF    AFG    1997 new   sp    m     4554      5
##  6 Afghanistan AF    AFG    1997 new   sp    m     5564      2
##  7 Afghanistan AF    AFG    1997 new   sp    m     65        0
##  8 Afghanistan AF    AFG    1997 new   sp    f     014       5
##  9 Afghanistan AF    AFG    1997 new   sp    f     1524     38
## 10 Afghanistan AF    AFG    1997 new   sp    f     2534     36
## # ... with 76,036 more rows
```

3 Missing Values

Changing the representation of a dataset brings up an important subtlety of missing values. Surprisingly, a value can be missing in one of two possible ways:

- **Explicitly**, i.e. flagged with NA.
- **Implicitly**, i.e. simply not present in the data.

Let's illustrate this idea with a very simple data set:

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

There are two missing values in this dataset:

- The return for the fourth quarter of 2015 is explicitly missing, because the cell where its value should be instead contains NA.
- The return for the first quarter of 2016 is implicitly missing, because it simply does not appear in the dataset.

The way that a dataset is represented can make implicit values explicit. For example, we can make the implicit missing value explicit by putting years in the columns:

```
stocks %>%
  pivot_wider(names_from = year, values_from = return)
```

```
## # A tibble: 4 x 3
##     qtr `2015` `2016`
##   <dbl>  <dbl>  <dbl>
## 1     1   1.88  NA
## 2     2   0.59   0.92
## 3     3   0.35   0.17
## 4     4  NA      2.66
```

Then we can turn this into a long dataset:

```
stocks %>%
  pivot_wider(names_from = year, values_from = return) %>%
  pivot_longer(
    cols = c(`2015`, `2016`),
    names_to = "year",
```

```
    values_to = "return"
    )
```

```
## # A tibble: 8 x 3
##     qtr year  return
##   <dbl> <chr>  <dbl>
## 1     1 2015    1.88
## 2     1 2016   NA
## 3     2 2015    0.59
## 4     2 2016    0.92
## 5     3 2015    0.35
## 6     3 2016    0.17
## 7     4 2015   NA
## 8     4 2016    2.66
```

If we didn't want the NA's to appear, we can set `values_drop_na = TRUE`:

```
stocks %>%
  pivot_wider(names_from = year, values_from = return) %>%
  pivot_longer(
    cols = c(`2015`, `2016`),
    names_to = "year",
    values_to = "return",
    values_drop_na = TRUE
    )
```

```
## # A tibble: 6 x 3
##     qtr year  return
##   <dbl> <chr>  <dbl>
## 1     1 2015    1.88
## 2     2 2015    0.59
## 3     2 2016    0.92
## 4     3 2015    0.35
## 5     3 2016    0.17
## 6     4 2016    2.66
```

Another important tool for making missing values explicit in tidy data is `complete()`:

```
stocks %>%
  complete(year, qtr)
```

```
## # A tibble: 8 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     1  NA
## 6  2016     2   0.92
## 7  2016     3   0.17
## 8  2016     4   2.66
```

The last missing data tool that we'll review is used for data where missing values indicate that the previous value should be carried forward:

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,           9,
  "Katherine Burke",  1,           4
)
```

You can fill in these missing values with `fill()`. It takes a set of columns where you want missing values to be replaced by the most recent non-missing value (sometimes called last observation carried forward).

```
treatment %>%
  fill(person)
```

```
## # A tibble: 4 x 3
##   person           treatment response
##   <chr>                <dbl>    <dbl>
## 1 Derrick Whitmore         1        7
## 2 Derrick Whitmore         2       10
## 3 Derrick Whitmore         3        9
## 4 Katherine Burke          1        4
```